An NL-based Foundation for Increased Traceability, Transparency, and Speed in Continuous Development of Information Systems

Bert de Brock
University of Groningen, Faculty of Economics and Business
PO Box 800, 9700 AV Groningen, The Netherlands
E.O.de.Brock@rug.nl

Abstract

The motivation for this paper comes from the problems in developing information systems that (a) the language and way of thinking of users usually differ (considerably) from the language and way of thinking of developers, (b) the user wishes are often unclear (at least initially), and (c) user wishes continuously change over time. These problems are all requirements-related and partly related to natural language (NL) use. Moreover, nowadays the 'times to market' should be shorter and shorter and environments are changing quicker and quicker. This asks for increased traceability, transparency, and development speed during the continuous development and evolution of information systems. Our principal results and contributions are: A novel, NL-based development path for functional requirements. This development path enables 'stepwise clarification' and 'stepwise specification' in order to tackle the problems mentioned above. It gradually goes from the informal natural language and way of thinking of users to the more formal language and way of thinking of developers (inputs, outputs, parameters, procedures, etc.). Some general (NL-)structures and forms for such a development path are studied. Moreover, an evolutionary development path for a whole system is presented. These development paths form a foundation for increased traceability, transparency, and development speed in the continuous development and evolution of information systems. Moreover, the nature of increments (extensions and adaptions) and their development are studied in detail.

Keywords: Continuous Development, Development Path, Natural Language, Stepwise Clarification, Stepwise Specification, Traceability, Transparency, Development Speed, Increment, (Conservative) Extension, Adaption, Agility.

1 Introduction

Some well-known problems during the development of information systems are that (a) the language and way of thinking of users usually differ (considerably) from the language and way of thinking of developers, (b) user wishes are often unclear (at least initially), and (c) the user wishes also change over time (a.k.a. 'requirements drift'), often due to internal and/or external changes. These problems are often important causes that software-development projects are not within budget, are not within time, and/or have inadequate functionality (hence failing on the basic project requirements). Moreover, nowadays the 'times to market' should be shorter and shorter and environments are changing quicker and quicker. In such a context of continuous development these problems ask for increased traceability, transparency, and development speed. We tackle (the consequences of) these requirements-

and NL-related problems by proposing a transparent and easily traceable development path for functional requirements that enables ‘stepwise clarification’ and ‘stepwise specification’. (We note that our focus is on the category of functional requirements, not on ‘non-functional’ requirements.) For more background on the traceability problem we refer to [GF94, Cle12].

The rest of the paper is organized as follows: Section 2 sketches our development path for functional requirements (FRs) and explains the basic concepts. The development path enables ‘stepwise clarification’ and ‘stepwise specification’. Section 3 gives a detailed example of the development path for a functional requirement. Section 4 presents some general structures and forms for such a development path. This core section clearly extends our previous work. Section 5 sketches an evolutionary development path for a whole system. Section 6 zooms in on such evolutionary development paths and emphasizes the nature of increments, in particular extensions and adaptions. These development paths form a foundation for increased traceability, transparency, and development speed in the continuous development and evolution of information systems.

2 Basic Concepts

We will sketch an NL-based development path for FRs. This development path is an extended and refined version of the one sketched in [Bro18a, Bro18b]. The path is straightforward and starts with the new notion of *user wishes* and then goes from *user stories* via *use cases* and their *system sequence diagrams* to a so-called *information machine* (model synthesis) and finally to a realization, an *information system*. We first explain these 6 notions.

Informally, a **user wish** (UW) is a ‘wish’, expressed in NL, of a (future) user which the system should be able to fulfil. E.g., for a student registration system for a university, a wish could simply be to ‘*Register a student*’. A user wish can originate from the prospective user, his/her boss, or another stakeholder, but is not yet very specific.

A **user story** (US), the next step in our ‘stepwise clarification’, is a user wish extended with the role of the intended user and the relevant parameters, e.g., the wish of an administrator to ‘*Register a student with a given name, address, and phone number*’. So, a user story is a more detailed specification than a user wish:

US = UW + user role + relevant parameters

Initially written user stories might need to be improved/refined/detailed/completed to better clarify what the system should do. According to [Luc16] user stories are popular as a method for representing requirements, especially in agile development environments, e.g., using a simple (but popular) template like

‘**As a** <role>**, I want to** <wish> [**so that** <benefit>]’

originating from [Coh04]. This template also contains an optional *benefit*-part.

A **use case** (UC) is a text in natural language that describes the sequence of steps in a typical usage of the system [Jac11, UC19], say to realize a US. A UC corresponds roughly to an *elementary business process*, a building block in business process engineering [Lar05]. For structuring purposes, larger UCs are often refined into smaller parts (sub-functions or ‘sub use cases’), usually factoring out duplicate sub-steps shared by other UCs [Lar05]. For a UC for our sample US we refer to Example 1 (to be found in Section 3).

UWs, USs, and UCs are all expressed in the natural language of the user (say English or Dutch). So, it is possible for the users (or domain experts) themselves to write them and/or validate them, maybe with the help of some (business) analyst.

A **system sequence diagram** (SSD) of a use case is a schematic ‘diagram’ emphasizing the interaction between the primary actor (user), the system (as a black box), and other actors (if any), including the messages (with their parameters) between them [Lar05]. The advantage of an SSD (as a kind of stylised UC) over a standard UC is that it makes the prospective inputs, state changes, and outputs of the system more explicit. An SSD clarifies the work to be done by the developer. In principle, there should be a one-to-one correspondence between the steps in the SSD and those in the UC.

SSDs are usually drawn as UML-diagrams (e.g., see [Lar05, SSD19]), but we only write down their bare essence (because that eases their analysis): We denote a basic step in an SSD as <Actor 1> → <Actor 2>: <Message>, meaning that <Actor 1> sends <Message> to <Actor 2>. For an SSD for our sample UC we refer to Example 1 again.

Given the desirable one-to-one correspondence between the SSD and the underlying UC, an SSD can be checked against that UC together with the user. If the UC and SSD don't match, the SSD might be wrong but it might also be that the UC is incomplete.

In order to formalise the informal requirements, we introduce the notion *information machine* (IM).
An **information machine** is a 5-tuple (I, O, S, G, T) consisting of:

1. a set I (of *inputs*)
2. a set O (of *outputs*)
3. a set S (of *states*)
4. a function G: I × S → O (the *output function*),
   mapping input-state pairs to the corresponding output
5. a function T: I × S → S (the *transition function*),
   mapping input-state pairs to the corresponding next state

The notation f: X → Y indicates that f is a function with domain X and its range being a subset of Y. The working of an information machine can be illustrated in a picture (with i ∈ I and s ∈ S):

$$i \rightarrow \boxed{s \longmapsto T(i,s)} \rightarrow G(i,s)$$

In words: Upon an input i, an information machine in state s produces the output G(i,s) and changes its internal state from s to T(i,s). So, both the output and the 'new' state depend on the input and the current state.

An information machine is equivalent to the notion of *data machine* in [Pie89] and to a (not necessarily finite) *Mealy machine* without a special start state; see [Mea55].

An IM is a kind of formal, mathematical *blueprint*: It is useful as an intermediate model, both as a clear, unambiguous semantic model of the user wishes regarding the functional requirements as well as a formal model of the system to be built.

We call the realization of an IM an **information system** (IS), i.e., 'an organized system for the collection, organization, storage and communication of information' [JV08, IS19]. An IM can be realized by different kinds of ISs. For instance, an IM can be realized by a human servant (say a clerk or monk), by an 'SQL servant' (i.e., a computer with SQL software), or by a 'Java servant' (i.e., a computer with Java software).

For a realization of user wishes in an object-oriented system you might use *methods*, and in a relational system you might use (stored) *procedures*. For the realization of our sample US in a relational (SQL-)system, for instance, we refer to Example 1.

3 Development Path for a Functional Requirement: An Example

Example 1 shows the complete (and straightforward**)** development path for a functional requirement, which starts with an initial user wish and then going from a user story via a use case and its system sequence diagram to an information machine and finally to a realization in an information system. The initial steps are NL-based. Example 1 also illustrates the notions of *system-generated values* and *compound transactions*.

Example 1: Development path for a functional requirement

This example shows the complete development path of the initial user wish *Register a student* (UW1). It goes from the user story US1 via the use case UC1 and its system sequence diagram SSD1 to the information machine IM and finally to a realization in the information system IS. Note that the (same) numbering of the artefacts increases (forward *and* backward) traceability.

Upon registration, a new student gets a system-generated student number in our sample university. This is mentioned in the UC and SSD, and worked out in the IM and IS. Below, the student name should be composed of the initials followed by the family name, and the address should be composed of street, postal code, and city.

The symbol "&" below denotes string concatenation, "Str" the set of character strings (over a certain alphabet), and "ℕ" denotes the set of natural numbers. For the IS we translate to a relational system and use (pseudo-)SQL. In SQL, parameter names are preceded by an "@".

UW1 Register a student

US1 As an administrator, I want to
Register a student with a given name, address, and phone number

UC1
1. The administrator (user) asks the system to
   Register a student with a given name, address, and phone number
2. The system uses the next unused student number as the new student number
3. The system registers the name, address, phone number, and student number
4. The system returns the assigned student number to the user
5. The system increases the next unused student number by 1

SSD1
1. User → System: RegisterStudent(<name>, <address>, <phone number>);
2. System → System: use the next unused student number as the new student number;
3. System → System: register the name, address, phone number, and student number;
4. System → User: “Assigned student number is ” <student number>;
5. System → System: increase the next unused student number by 1

IM

State space: Consists of an integer-valued component (NUSN: ℕ) and
a set-valued component (STUD), with 4 properties (+ their value sets):
NAME: Str, ADDR: Str, PHNR: Str, SNR: ℕ

Inputs: { RegisterStudent(n,a,p) | n ∈ Str and a ∈ Str and p ∈ Str }
For input i = RegisterStudent(n,a,p) and state s:

Outputs: G(i,s) = “Assigned student number is ” & s(NUSN)

Transitions: T(i,s) is the state in which the new NUSN-value is s(NUSN) + 1 and
the new STUD-value is s(STUD) ∪ {t1}; here t1 is the function defined by
t1(SNR) = s(NUSN), t1(NAME) = n, t1(ADDR) = a, and t1(PHNR) = p.
We note that this is a *compound transaction*, because 2 components change

IS `CREATE DATABASE StudentRegistration` */* Student registration database*

Data structure
```
CREATE TABLE  STUD
(NAME  VARCHAR    NOT NULL,
 ADDR  VARCHAR    NOT NULL,
 PHNR  VARCHAR    NOT NULL,
 SNR   INTEGER    NOT NULL)

CREATE VARIABLE  NUSN  AS INTEGER
```

Procedure implementing the user wish UW1 (*Register a student*)
```
CREATE PROC RegisterStudent @name VARCHAR, @addr VARCHAR,
             @phnr VARCHAR, @output VARCHAR OUTPUT   AS
BEGIN INSERT INTO STUD (NAME, ADDR, PHNR, SNR)
                 VALUES (@name, @addr, @phnr, NUSN);
     SELECT @output = ‶Assigned student number is ″ & NUSN;
     UPDATE NUSN SET NUSN = NUSN + 1
END
```

4 Some General Structures

In Example 1 we deliberately translated the initial user wish (*Register a student*) into a system input with a recognizable name (*RegisterStudent*) in the SSD and in the IM, together with the relevant parameters (as introduced in the US). And we also translated it into an IS-procedure with the same name (*RegisterStudent*) and the same parameters, plus a parameter for the output. This naming policy clearly increases traceability!

Note that Step 1 in the UC/SSD gives a clue for the *heading* of the IS-procedure and that the other steps in the UC/SSD are clearly reflected in the *body* of that procedure.

The general structure in the development path in Example 1 is as follows:

UW The user wish has the form: <(action) verb> **a** <noun (phrase)>

We note that something similar holds for many other NLs (e.g., Dutch, French, German).

The more general form is: <(action) verb> <indefinite article> <noun (phrase)>

US The user story consists of the user wish extended with the intended user role and the relevant parameters. It has the form:

As a <role>**, I want to** <user wish> **with a given** <parameter list>

where <user wish> is of the form: <(action) verb> **a** <noun (phrase)>

UC The first step in the use case subsequently has the form:

The <role> (**user**) **asks the system to** <user wish> **with a given** <par. list>

where all ingredients are taken directly from the US. The following steps in the use case informally describe what the system has to do (but not *how* to do it). Usually, the system must also return an output to the user.

SSD An SSD is a kind of stylised UC. The first step in the SSD then has the form:

User → System: ActionName**(**<parameter list>**)**

where (for traceability reasons) ActionName can be chosen as the concatenation of <(action) verb> and <noun (phrase)> from the original <user wish>. So, the first step of the SSD follows directly from the US (or from the first step in the UC).

IM We can read from that first step of the SSD that ActionName**(**<parameter list>**)** must be an IM-input for all possible value combinations for <parameter list>. For each such input i we must formally specify (in detail) the output G(i,s) and next state T(i,s) for each state s, in order to express the desired effect precisely.

IS For the realization of a user wish one can use a *method* in an object-oriented system or a (stored) *procedure* in a relational system. As the name of the method/procedure you can use that same ActionName (again for traceability reasons) and inherit its parameters from the parameter list in the IM (plus maybe an output parameter). The steps in the UC/SSD (except the first one) are clearly reflected in (the structure of) the *body* of the method/procedure in the IS, where the output function and the transition function of the IM provide the details.

Table 1 roughly summarizes those subsequent, related grammatical forms (where α denotes the action verb and β the noun phrase in the original user wish):

Table 1: Summary of the Relationship between the Subsequent Grammatical Forms

UW	α **a** β
US	**As a** <role>**, I want to** α **a** β **with a given** <parameter list>
UC	First step: **The** <role> (**user**) **asks the system to** α **a** β **with a given** <par. list>
SSD	First step: **User → System:** $\alpha\beta$**(**<parameter list>**)** where User is a <role>
IM	Inputs: $\alpha\beta$**(**<parameter list>**)** for all possible value combinations of <par. list>
IS	Method/procedure $\alpha\beta$ with <parameter list> (plus maybe an output parameter) of which the body stems from the UC/SSD-structure and the IM-details

A user wish or user story is often the starting point of a development path (e.g., in agile development) and is usually expressed using an *action verb* and a *noun phrase*. From Table 1 we can conclude that the naming policy in Example 1 can be applied in general:

A user wish of the form $\alpha\gamma\beta$ (where α denotes an action verb, γ the indefinite article, and β a noun phrase) can be transformed into a method or procedure named $\alpha\beta$ in the final information system, and the parameter list introduced in the user story can be transformed into a similar parameter list in that same method/procedure in the final IS. Vice versa, from the name of a method or procedure in the IS we can infer the original user wish!

So, this simple naming policy provides **bi-directional traceability**, all the way from the original user wish to the final software code! This generally applicable naming policy also provides **transparency** in the whole lifecycle of a functional requirement. And thanks to the straightforward relationship between the separate steps in the development paths, our approach increases the **development speed** as well!

Regarding the action verbs: We note that there are 4 general basic functions applicable to data in a system, in the literature known as CRUD (Create, Read, Update, and Delete; e.g., see [Mar83, CRU19]): One can add data to the system (Create), only 'look' at data in the system (Read), change data in the system (Update), or remove data from the system (Delete). The action verb in the UW should preferably indicate this already, i.e., indicate what the FR category is. Table 2 presents some relevant action verbs.

Table 2: Some Basic FR Categories: The CRUD-Verbs and some Alternatively Used Action Verbs

CRUD	**Some alternatively used action verbs**
Create	Register, Add, Enter
Read	Retrieve, View, See, Search
Update	Refresh, Change, Modify, Edit, Alter, Adapt, Replace, Rename
Delete	Remove, Destroy

As mentioned before, UWs, USs, and UCs are expressed in the (natural) language of the user (say English or Dutch). So, it is possible for the users (or domain experts) themselves to write them and/or validate them. They might use action verbs other than the ones already mentioned in Table 2. A (business) analyst may help to find out the basic FR category, i.e., clarify whether a user wish is a C-, R-, U-, or D-wish.

Regarding the noun phrases: They heavily depend on the application area at hand. In other words: they are *domain specific*. E.g., typical noun phrases in a university environment are Student, Course, Lecturer, Exam, Grading, Course Enrolment, Exam Enrolment. In combination with the action verbs we get action names like *RegisterStudent*, *RetrieveStudent*, *UpdateStudent*, *RemoveStudent*, *RegisterCourse*, *RetrieveCourse*, etc. Usually (but not always), each such combination makes sense.

Example 1 (and its generalization summarized in Table 1) clearly concerns a Create-case. The other 3 CRUD-cases can have similar forms. For instance, an obvious Read-case could be to retrieve the info of a particular student, indicated by his/her student number. Similarly, an obvious Delete-case could be to remove a particular student (indicated by his/her student number). And an Update-case could be to update some of the parameter-values of a particular student (indicated by his/her student number). In Table 3 below we denote this sub-list of the parameters by <par. sub-list>.

In this way, the general structure extracted from Example 1 for the Create-case applies similarly to these other 3 CRUD-cases. This leads to the general CRUD-forms indicated below (for the US/UC-part respectively SSD/IM-part). The form in the IS-part is the same as in the SSD/IM-part, plus maybe an additional output parameter.

Table 3: Grammatical Forms for each CRUD-Case (i.e., Basic FR Category)

	Verb α	**Form in the US/UC-part**	**Form in the SSD/IM-part**
C	Create/Register/…	α **a** β **with a given** <parameter list>	αβ**(**<parameter list>**)**
R	Read/Retrieve/…	α **the** β **with a given** <ID>	αβ**(**<ID>**)**
U	Update/Refresh/…	α <par. sub-list> **of the** β **with a given** <ID>	αβ**(**<ID>, <par. sub-list>**)**
D	Delete/Remove/…	α **the** β **with a given** <ID>	αβ**(**<ID>**)**

In [Bro18b] general development *patterns* for each of the four CRUD cases are worked out. Therefore, given such general development patterns and because many user wishes are relatively simple, not all user wishes need to be worked out in all detail each time [Lar05].

5 Development Path of a System

Until now we looked at the development path of a single user wish only. What about the development path of a whole system? You could develop a system in one go, including 'all' functionality that is needed (the 'waterfall method' or 'big bang'). However, in practice systems are really sophisticated, supporting many (subtle) user wishes, resulting in a (very) large IM and IS. Moreover, in practice such systems are often under continuous development ('under construction'), just as a city for instance.

Instead, you may start with a simple, small version of the system and extend/adapt it in several steps to larger and more sophisticated versions, a.k.a. *incremental* development. The idea of increments introduces flexibility in the development process.

The proposed development path for a single user wish (UW→US→UC→SSD) in combination with incremental development leads to the following scheme (where the arrows indicate what is input for what):

$$\begin{tikzcd}
\text{UWs} & \text{UW} \arrow[d] & \cdots & \text{UW} \arrow[d] & \vdots & \text{UW} \arrow[d] & \cdots & \text{UW} \arrow[d] & \vdots \\
\text{USs} & \text{US} \arrow[d] & \cdots & \text{US} \arrow[d] & \vdots & \text{US} \arrow[d] & \cdots & \text{US} \arrow[d] & \vdots \\
\text{UCs} & \text{UC} \arrow[d] & \cdots & \text{UC} \arrow[d] & \vdots & \text{UC} \arrow[d] & \cdots & \text{UC} \arrow[d] & \vdots \\
\text{SSDs} & \text{SSD} \arrow[dr] & \cdots & \text{SSD} \arrow[dl] & \vdots & \text{SSD} \arrow[dr] & \cdots & \text{SSD} \arrow[dl] & \vdots \\
\text{IM} & & \text{IMv1} \arrow[rrrr] \arrow[d] & & & & \text{IMv2} \arrow[rr] \arrow[d] & & \cdots \\
\text{IS} & & \text{ISv1} \arrow[rrrr] & & & & \text{ISv2} \arrow[rr] & & \cdots
\end{tikzcd}$$

Figure 1: Envisaged Development Path for a System (Incrementally)

Figure 1 in words: Via one or more UWs, USs, UCs, and their corresponding SSDs (using stepwise clarification/specification), plus the previous version of the IM, you can define an initial (resp. next) version of the IM and, based on the IM (plus the previous IS-version), you can define an initial (resp. next) version of the IS.

Altogether, this represents an *evolutionary* systems development path. And since the stepwise clarification/specification applies to individual user wishes, it scales up well.

As mentioned in [Bro18a], one cycle might contain only a few UWs, USs, UCs and their corresponding SSDs, or maybe even only *one* UW, US, UC and SSD. Or, maybe even *less than one* full UC: In agile development of functional requirements a simple 'core' scenario (or 'Main Success Scenario') of a - maybe yet unclear - 'full' UC might be developed first, followed by 'fuller' versions in subsequent cycles (see [Lar05]). So, previously developed UW/US/UC/SSD-series might be adapted too.

Cycles are even very short in case of *daily/nightly builds* (see [Db19]) and *continuous integration (*see [Boo98]).

6 Evolutionary Development and the Nature of Increments

We sketched and illustrated a development path for individual functional requirements (Section 2), presented some general structures and forms for such a development path (Section 4), and sketched an evolutionary development path for a whole system (Section 5). In the current section we zoom in on such evolutionary development paths and emphasize the nature and consequences of increments (extensions and/or adaptions).

When an increment is introduced while the system is already in use (so, a 'running' system), you must also specify what should happen with the 'current' state. This is necessary when applying *development* during *deployment*. It might be that the new state space S' includes all states in the old state space S (i.e., $S \subseteq S'$), and hence also the 'current' state. But it could also be that the structure of the new states differs from the structure of the old states, in which case it might even be that $S \cap S' = \emptyset$. In that case we must specify how the 'current' state should change to 'fit' in the new state space S'.

An increment can be a conservative extension, meaning that there is new functionality but that existing ingredients stay the same. In terms of an IM: There are only new inputs (and outputs) but the state space of the IM stays the same and, restricted to the existing inputs, the output function and transition function stay the same too. So, with a conservative extension, you only have to specify the output G(i,s) and next state T(i,s) for each new input i (and each state s). In case of a ‘running’ system, i.e., a system already in use, the ‘current’ state can stay the same as well.

Another known possibility is that an increment is an adaption, meaning that essentially there is no new functionality but that existing ingredients change.

Often an increment is a combination of an extension and an adaption: There is new functionality and, usually as a consequence, some existing ingredients change as well.

Conclusions

The paper introduced an NL-based, transparent and easily traceable development path for functional requirements that goes all the way from initial user wishes (such as *Register a student*) to a software realization (e.g., a *method* in an object-oriented system or a (stored) *procedure* in a relational system). The development path for a whole system is sketched as well.

We showed that the development paths are straightforward and increase transparency and traceability of functional requirements. In our daily practice we experience that they enable extensions and adaptions in a flexible way, enable incremental and agile development, and speed up the development process. In this way, the paper solved some challenges with functional requirements expressed in NL.

References

[GF94] O.C.Z. Gotel, C.W. Finkelstein: An analysis of the requirements traceability problem. Requirements Engineering, 94-101 (1994).

[Cle12] J. Cleland-Huang et al: Software and Systems Traceability. Springer (2012).

[Bro18a] E.O. de Brock: Towards a Theory about Continuous Requirements Engineering for Information Systems. In: CRE Workshop, REFSQ (2018).

[Bro18b] E.O. de Brock: Towards pattern-driven requirements engineering: Development patterns for functional requirements. In: MoDRE Workshop, RE-conference (2018).

[Luc16] G. Lucassen, F. Dalpiaz, J. M. E. M. van der Werf, S. Brinkkemper: The Use and Effectiveness of User Stories in Practice. In: Proc. of the International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ), 205–222 (2016).

[Coh04] M. Cohn: User Stories Applied: For Agile Software Development. Addison Wesley (2004).

[Jac11] I. Jacobson et al: Use Case 2.0: The Guide to Succeeding with Use Cases. Ivar Jacobson International (2011).

[UC19] https://en.wikipedia.org/wiki/Use_case

[Lar05] C. Larman: Applying UML and patterns. Pearson Education (2005).

[SSD19] https://en.wikipedia.org/wiki/System_sequence_diagram

[Pie89] F.T.A.M. Pieper: Data machines and interfaces. PhD thesis, TU Eindhoven (1989).

[Mea55] G.H. Mealy: A Method for Synthesizing Sequential Circuits. Bell System Technical Journal, 1045–1079 (1955).

[JV08] L. Jessup, J. Valacich: Information systems today. Pearson (2008).

[IS19] https://en.wikipedia.org/wiki/Information_system

[Mar83] J. Martin: Managing the Data-base Environment. Prentice Hall (1983).

[CRU19] https://en.wikipedia.org/wiki/Create,_read,_update_and_delete

[Db19] https://en.wikipedia.org/wiki/Daily_build

[Boo98] G. Booch: Object-oriented analysis and design with applications. Addison Wesley (1998).

All links were last accessed on 2019/01/11